Jeliot 3 Intermediate Code Specifications MCode

Niko Myller and Andrés Moreno García

31st May 2004

Contents

Chapter 1

Introduction

This document contains the specification for the MCode, intermediate language code used in Jeliot 3 program visualization system.

1.1 Purpose

This document main intention is to lay a solid ground for MCode, so future modifications, additions and queries to it will have a clear reference within this document.

1.2 Scope

This document attaches to Jeliot 3. MCode was developed in order to provide the execution information to an interpreter, which manages the visualization scene. The interpreted language is Java and the interpretation is done by DynamicJava. DynamicJava is a Java interpreter written in Java. Java and DynamicJava imposes some properties to the MCode. Furthermore, the original target users (programming novices) and their programs, also delimits the Java features that are currently supported by the MCode, such features as threads and reflection utilities are not supported. More on this will be explained in the following chapter.

1.3 Definitions, acronyms and abbreviations

Here it is included some definitions of used word to help its comprehension.

1.4 References

Interested readers should point to the master theses written by Niko Myller and Andrés Moreno, both can be found at Jeliot 3's webpage: `http://cs.joensuu.fi/jeliot/`. Niko's thesis (Myller, 2004) describes Jeliot 3 system and its implementation. Andrés' thesis (Moreno, 2004) explains further the decisions that shaped this intermediate code and compares it with different solutions, describing a taxonomy.

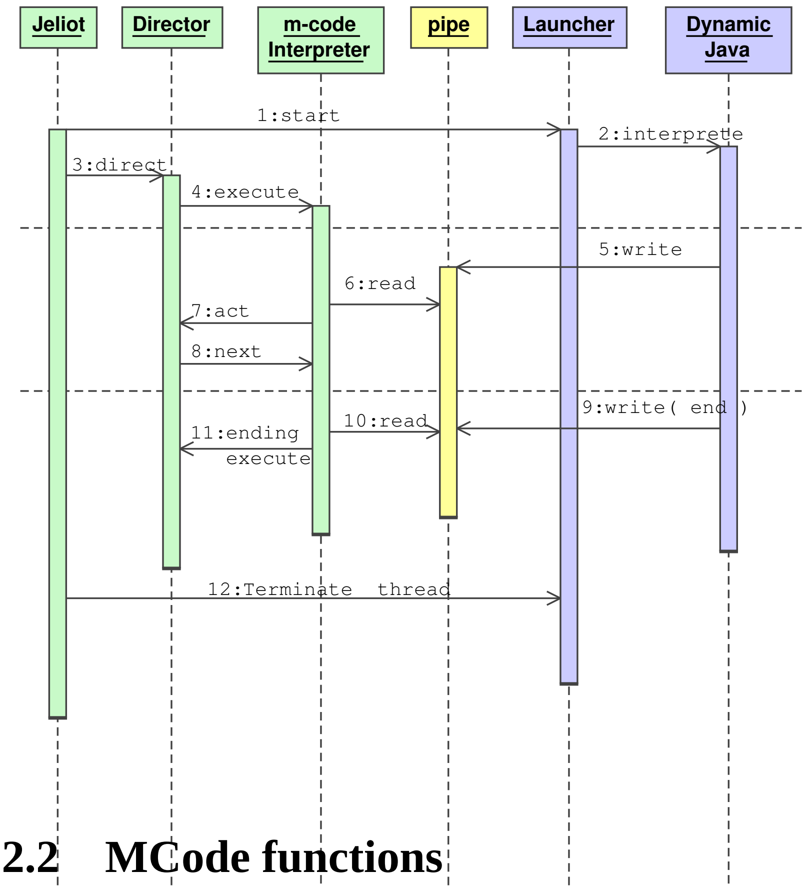
1.5 Overview

This language will be used to transfer evaluation information between Dynamic Java and Director class of Jeliot 3. The information flows to one direction, from Dynamic Java to Director (with a possible exception of the Input statements). Section 2 will provide a background, introducing Jeliot 3 system and some features of MCode.

Chapter 2

Overall Description

2.1 Jeliot 3 System Structure



2.2 MCode functions

MCode is the intermediate language used to communicate the visualization engine and the Java interpreter (DynamicJava). It mainly flows from DynamicJavato the Director. The information that carries along within is not only the information about modified variables and modified method stacks, but also the operations that produce such changes. and the results of those operations. This is the biggest difference from normal compiler intermediate codes; they only indicate what operations to perform to the assembler. In our case all operations are performed and the every result is sent to the Director.

2.3 User Characteristics

This document is addressed to the following people:

Developer of Visualization Systems This document may provide ideas and solutions to developers of new visualization systems, as well as provides of an existing solution that can be merged into their ongoing projects.

Maintainer of Jeliot 3 Future developers and maintainers of Jeliot 3 should refer to this document when modifying its source code, specially those files referring to DynamicJava and Jeliot 3's interpreter. They should incorporate here all changes done to MCode to provide a state of the art document as well.

2.4 Features and Constraints

While trying to produce the more generic intermediate code some features and constraints where placed due to several reasons. The most important ones are the following ones:

Java Being Java the language of choice, MCode supports most of its characteristics and abilities. It is object oriented, so things as method calls, objects and, to some extend, inheritance are supported.

DynamicJava Because of using DynamicJava as an interpreter from which to extract the execution information, some of the specified ordered sequences of instructions may be too constrained to the particularity of DynamicJava. However that is not the case for the most of instructions explained here.

Targeted audience MCode was designed to support the development of MCode, thus it was important to address their specific problems: MCode has a great detail in the evaluation of expressions and the normal flow of program, so novices will grasp the programming fundamental issues. Meanwhile, more advanced features are not within the scope of the MCode and have not been implemented. Those features include threads, reflection utilities and exception handling.

2.5 Dependencies

Right now MCode is only produced through the modified version of DynamicJava included in the distribution of Jeliot 3. However it is up to anyone to create a new high-level interpreter to produce its own MCode to be run by Jeliot 3's MCode interpreter or by one they develop. There is also the possibility to store the MCode a single text file and be retrieved later by a MCode interpreter to produce visualization orders. However, this configuration would not allow INPUT/OUTPUT operations as they provide information to the evaluation needed to carry on with following instructions.

Chapter 3

MCode Language

3.1 Introduction

MCode is defined to be an interpreted line by line and each line can be divided in the smaller pieces or tokens. Here we define all the commands that are part of MCode. First of all, we will introduce the notation used in this specification. as it is not standard. In our notation each token is now separated by single '`§`' character, this token will not be shown here instead a blank space will be used to help readability. Each token is first introduced as an English name. Later a sample instruction is given, consisting of the different tokens that build that particular instruction. If the token is in big letters it is a preserved word. If it is in small letters it will be replaced by variable value or integer. Table 3.1 shows the abbreviations used in the specification to refer to some common tokens:

The usual MCode sentence will consist of:

`Expression/Statement code` A shortcut for every Java statement or expression is used: e.g. AE stands for Add Expression. The chosen names are heavily related to the ones used in DynamicJava.

`Reference` Every Expression/Statement sentence is identified by a number. This way nested statements and expressions can be formed up from previous m-code sentences.

`Related References` Most of the m-code sentences refer to previous m-code sentences. One Add Expression will refer to the references of both sides of expression. Flow-control statements will refer to a condition expression, and so on.

`Value` Most sentences will return the value resulting from the executing of an expression. If it is a flow control statement it will return a Boolean value indicating the result of the condition.

Name	Abbreviation	Meaning
Instruction Reference	`ir`	Used to keep track of instructions used in the past.
Left Instruction Reference	`lir`	A reference to a previous instruction used as the left operand.
Right Instruction Reference	`rir`	A reference to a previous instruction used as the right operand.
Instruction Counter	`ic`	Counter of expressions, making instructions unique and easily referenciable
Type of instruction	`ti`	Refer to some m-code instruction (e.g. `ADD`)
Location	`lo`	Variable that holds the location of the expression in the source code, defined by “beginning of line”, “beginning of column”, “end of line”, “end of column”.

Table 3.1: Token Abbreviations.

`Type` Every expression that has a result must specify its type.

`Location` This contains the location of the expression in the original source code file.

3.2 Grammar

The following grammar expresses the syntactic properties of the MCode. It describes how to get a sintactically correct MCode program, that can be parsed by the interpreter provided Jeliot 3 to produce the right visualizations. This grammar has been obtained by modifying the Java BNF sintactic grammar.

Tokens within "<" and ">" are Non-Terminal symbols, capitalized words are Terminal symbols, basically MCode commands; you will find a more detailed description of them in the follwing section."?" means that the preceeding symbol is optional, and "|" separates the different possibilities.

<program> ::= <classes info> <static method call> `END`[1]

<classes info> ::= <class info>
| <classes info> <class info>

<class info> ::= `CLASS METHOD CONSTRUCTOR FIELD`? `END_CLASS`

[1]The beginning Static Method Call it is supposed to call the Main method of the program

```
<static method call> ::= SMC <method call info> <method body> SMCC

<object method call> ::= OMC <method call info> <method body> OMCC

<method call info> ::= PARAMETER MD <parameters info>?

<parameters info> ::= <parameter info>
                    | <parameters info> <parameter info>

<parameter info> ::= BEGIN <expression> P

<method body> ::= <block statements>

<block statements> ::= <block statement>
                     | <block statements> <block statement>

<block statement> ::= <variable declaration>
                    | <statement>

<scoped block> ::= SCOPE 1 <block statements> SCOPE 0

<statement> ::= <simple statement>
              | <if statement>
              | <for statement>
              | <while statement>

<simple statement> ::= <scoped block>
                     | <statement expression>
                     | <switch statement>
                     | <do statement>
                     | <continue statement>
                     | <break statement>
                     | <return statement>

<statement expression> ::= <assignment>
                         | <pre expression>
                         | <post expression>
                         | <method call>
                         | <class allocation>

<if statement> ::= <expression> IFT <statement>?
                 | <expression> IFTE <statement>?
```

<while statement> ::= <expression> `WHI` <statement>?

<switch statement> ::= `SWITCHB` <expression> `SWITCHBF` <block>? `SWITCH`

<for statement> ::= `SCOPE 1` <variable declarations> <expression> `FOR` <statement>? `SCOPE 0`

<do statement> ::= <block> <expression> `DO`

<break statement> ::= `BREAK`

<break statement> ::= `CONTINUE`

<return statement> ::= <expression>? `RETURN`

<expression> ::= <assignment>
| <binary expression>
| <unary expression>
| <primary>

<assignment> ::= `BEGIN` <expression> `TO` <left hand side>

<left hand side> ::= <identifier>
| <field access>
| <array access>

<binary expression> ::= `BEGIN LEFT` <expression> `RIGHT` <expression> <binary operator>

<unary expression> ::= `BEGIN` <expression> <unary operator>

<primary> ::= <literal>
| <field access>
| <array access>
| <class allocation>
| <array allocation>
| <method call>

<literal> ::= `L`

<identifier> ::= `QN`

<variable declaration> ::= `VD`

<field access> ::= <primary> <identifier> OFA

<array access> ::= <identifier> <dim expressions> AAC

```
<dim expressions> ::= <expression> |
                      <dim expressions> <expression>


<method call> ::= <static method call>
                | <object method call>


<array allocation> ::= <dims expressions> AA

<class allocation> ::= SA <method call info> <method body> SAC


<variable declarations> ::=<variable declaration>
                          | <variable declarations> <variable declaration>


<pre expression> ::= <identifier> PRIE | <identifier> PRDE

<post expression> ::= <identifer> PIE | <identifier> PDE


<unary operator> ::= PLUS | MINUS | COMP | NO


<binary operator> ::=AE | SE | ME | DE | RE
                   | BITOR | BITXOR | BITAND
                   | LSHIFT| RSHIFT | URSHIFT
                   | XOR | AND | OR | EE | NE
                   | GT | LE | LQE | GQT
```

3.3 Constants

Table 3.2 contains the constants used in the implementation of MCode to support portability and maintainability.

3.4 Auxiliary Instructions

Some auxiliary instructions are defined in order to ease the interpretation of the m-code. These instructions are not related to any particular Java construct and are used by those that need them.

Constant	Meaning
`DELIM`	Used to separate tokens on a single instruction. They have to be explicitly added to the m-code instruction
`LOC_DELIM`	Used to separate the different coordinates that locate some source code.
`UNKNOWN`	When some variable value cannot be accessed it is given an `UNKNOWN` value.
`NO_REFERENCE`	??
`REFERENCE`	??
`NOT_FINAL`	??
`FINAL`	??
`TRUE`	String that represent the `TRUE` Boolean value in the working environment.
`FALSE`	String that represent the `FALSE` Boolean value in the working environment.

Table 3.2: Constants.

3.4.1 BEGIN

```
BEGIN ti ir loc
```

Instruction used to mark the beginning of those instructions that admit instructions to be encapsulated within them. Those instructions are Assignment, Return, Parameter, Array Access and all unary and binary operations; and they are referred through it. The referred instructions get they `ir` assigned in the `BEGIN` instruction.

3.4.2 LEFT and RIGHT

```
LEFT/RIGHT ir
```

These instructions are similar to `BEGIN`. Both of them are used to mark the beginning of the `left`/`right` side of a binary operation. The `ir` obeys the same reason than in `BEGIN`, “reserves” the reference number for the following instruction.

3.4.3 TO

```
TO ir
```

This instruction is used in assignments and reflects the movement of the value to the left hand side of the assignment. The `ir` points to the qualified name that will hold the value. As said before, this instruction is used in assignment and more concretely in Assignment, Variable Declaration (those with initializer) and Compound Assignment.

3.4.4 ERROR

```
ERROR errorMessage loc
```

Parser and execution errors are reported to the visualization engine with the `ERROR` instruction. The `errorMessage` is a string that can contain `HTML` and it is what will be visualized.

3.4.5 END

```
END
```

END is produced at the end of the MCode program and indicates the visualization to terminate.

3.4.6 SCOPE

```
SCOPE 1/0
```

New blocks of Java code are delimited in MCode through `SCOPE`. This instruction second token indicates whether it is opening a new one (`1`) or closing one (`0`).

3.4.7 CONSCN (Constructor Call Number)

```
CONSCN ir
```

CAUTION! This instruction is implementation specific and the need for the usage of this instruction depends on the implementation of the interpreter. If your interpreter traverses the tree in the right order there is no need to use the `CONSCN`. The usage of the instruction is next described in DynamicJava.

This instruction is created because in DynamicJava the super method calls in the beginning of the constructor are handled before the actual constructor invocation and thus the information is not extracted in the correct order. The constructor call number is used to correct the order so that during the simple allocation visit in EvaluationVisitor the constructor call number is send for the first time. When all the super method calls are finished and the constructor is really invoked the constructor call number is printed out again. The MCode interpreter collects all the commands between the corresponding constructor call numbers and executes them after the constructor is really invoked that is two lines after the second constructor call number is read. However this instruction is due to DynamicJava drawbacks. If your interpreter traverses the tree in a meaningful way, you should not use the CONSCN instruction.

3.5 Statements

3.5.1 A (Assignment)

```
A ir rir lir value type loc
```

The assignment instruction is composed by its own reference (`ir`) and the references to the left and right hand sides (`lir`, `rir`). Furthermore it contains the assigned value and its type.

It is worth to mention that compound assignments are decomposed into the operation and a simple assignment.

3.5.2 VD (Variable Declaration)

```
VD name NO_REFERENCE/ir value type FINAL/NOT_FINAL
loc
```

When declaring a variable the corresponding the MCode instruction needs to be complemented with its name, value, type and the modifier (`FINAL` or `NOT_FINAL`). Instruction reference (`ir`) is given if the variable has an initializer otherwise a `NO_REFERENCE` value is written.

3.6 Binary Operations

```
binaryCode ir rir lir value type loc
```

Binary instructions are composed by its `binaryCode`, its own reference (`ir`) and the references to the left and right sides of the expression (`lir`, `rir`). Furthermore, it contains the assigned `value` and its `type`. The `binaryCode` can take any of the values shown in Table 3.3

3.7 Unary Operations

```
unaryCode ir reference value type loc
```

As with binary operations the unary instructions (`unaryCode`) take the similar tokens. The only difference is that there is only one `reference` to another instruction. The `value`, `type` and `loc` maintain the same meaning. As before there are several

Table 3.3: Binary Operators that can be assigned to the `binaryCode`.

Boolean operators	MCode	Arithmetic operators	MCode	Bitwise operators	MCode
AND &&	`AND`	ADD +	`AE`	AND &	`BITAND`
OR |	`OR`	SUBTRACT -	`SE`	OR |	`BITOR`
XOR ^	`XOR`	MULTIPLY *	`ME`	XOR ^	`BITXOR`
LESSER THAN <	`LE`	DIVIDE /	`DE`	LEFT SHIFT <<	`LSHIFT`
GREATER THAN >	`GT`	REMAINDER %	`RE`	RIGHT SHIFT >>	`RSHIFT`
EQUAL ==	`EE`			UNSIGNED SHIFT >>>	`USHIFT`
NOT EQUAL !=	`NE`				
LESSER THAN OR EQUAL TO >=	`LQE`				
GREATER THAN OR EQUAL TO =<	`GQT`				

Java unary operators that can be assigned to the `unaryCode`, all of them are listed in Table 3.4.

3.8 Literal constant and variable access

3.8.1 Qualified Name

```
QN ir name value type
```

Qualified names are all the local variables. MCode instruction contains the reference (`ir`) of the instruction and the name value and type of the qualified name. If the variable is not initialized value will be `UNKNOWN`.

3.8.2 Literal

```
L ir value type loc
```

Literals are the constants values in the source code (e.g. `3` is one integer literal and “3”

Table 3.4: Unary Operators that can be assigned to the `unaryCode`.

Boolean Operators	MCode	Arithmetic operators	MCode	Bitwise operators	MCode
NOT !	NO	POSTINCREMENT ++	PIE	COMPLEMENT ~	COMP
		POSTDECREMENT --	PDE		
		PREINCREMENT ++	PRIE		
		PREDECREMENT --	PRDE		
		PLUS +	PLUS		
		MINUS -	MINUS		

is one string literal). The MCode instruction contains all the information needed for the visualization as well as its reference (`ir`), `value`, `type` and `loc`.

3.9 Control Structures

3.9.1 If Statements

```
IFT/IFTE condition value loc
```

There are two possible instructions for an "If" statement. `IFT` is printed if there is not an else statement or `IFTE` if there is an else statement. The composition, however, is similar. The `condition` is the reference to the instruction that evaluate the condition. The `value` holds the result of the evaluated condition and will tell which branch execution is following. The `loc` as usual contains the code location.

3.9.2 While For and Do-While Statements

```
WHI/FOR/DO condition value round loc
```

These statements produce a similar to the previous one. They only differ in the round token. This token holds the number of iterations the loop has made.

3.9.3 Switch

Three MCode instructions are related to the switch statement.

```
SWITCHB loc
```

Switch statement interpretation begins with this instruction. The location of the whole switch block is given as the parameter.

```
SWIBF selector ir loc
```

The `SWIBF` instruction is written when one of the cases is selected as the matching case. The selector gives a reference to the selector expression. The instruction reference gives the reference for the case expression. If the default case is selected then the instruction reference is set to `-1`. The location explains the location of the found case block.

```
SWITCH loc
```

This statement is used when the switch statement is exited if the break instruction (`BR`) is not given. The parameter for `SWITCH` instruction is the location of the whole switch block.

3.9.4 Break and Continue

```
BR/CONT statement loc
```

Break and continue asserts instructions only specify which statement they are in. The allowed values for `statement` are `WHI`, `FOR`, `DO` or `SWITCH`.

3.10 Input and Output

3.10.1 Input

```
INPUT ir type loc
```

The `INPUT` instruction indicates the visualization engine to produce some data of the type specified and return it to Java interpreter. This data will be written in a dedicated pipe that connects both sides. The next instruction indicates the obtained value from the pipe.

```
INPUTTED counter value type loc
```

3.10.2 Output

```
OUTPUT ir value type breakLine loc
```

`OUTPUT` instruction is the resulting one of a call to any of the Output methods provided by Jeliot 3 or System.out.print and System.out.println. They only accept one argument, and it is reflected in the instruction by its value and its type. A flag (`breakline` is added to indicate wheter to break the line (value `1`) or not (value `0`) in the output.

3.11 Array Handling

3.11.1 Array Allocation

```
AA dimension dimensionsReferences dimensionsSizes
loc
```

Array allocation instruction is a complicated one, as it carries a lot of information about the array. As usual a `ir` is provided. Following the `arrayHashCode` of the object created to allocate it. It can be any other number that identify one-to-one the allocated object on the interpretation. The `type` contains the type of the array components. The `dimensionReferences` is a comma separated list with the references to the instructions that evaluated the sizes' expressions of the array. Finally, the `dimensionsSize` is another comma separated list where each element is the size of a dimension. An expression "`new Integer[4][5]`" will produce a following line of MCode "`AA ir 456744 Integer 2 ir1,ir2 4,5 loc`". Where `ir1` and `ir2` must be references to the literal instructions of "4" and "5" respectively.

3.11.2 Array Access

```
AAC ir arrayNameReference deep cellReferences
cellNumbers value type loc
```

Array accesses instructions consist on different tokens. The common ones (`ir`, `value`, `type` and `loc`) are also present. But we can find very specific ones.

The `arrayReference` points to the instruction produced when visiting an array name, normally a qualified name. deep refers to the level of deepness of the array access. This is useful for multidimensional arrays when they are not accessed till the last level, the one holding the data value. The `cellReferences` and the `cellNumber` meet the same purpose than `dimensionReferences` and `dimensionsSizes` in array allocation (`AA`) instruction. The `cellReferences` points to the instructions that evaluated the value of each cell pointer. The `cellNumbers` are the actual values of the cell access. Both of them are presented as a comma separated list. For example in “`array[3][5]`”, `cellReferences` will point to the literal instruction of “3” and “5” and `cellNumbers` will contain “3,5”.

3.11.3 Array Length

```
AL objectCounter "length" value type loc
```

3.12 Object Oriented

This section contains the instruction related to methods and object oriented programming. Here we will just summarize its properties and main arguments. However, following chapters will explain you how to glue the different instructions together to form meaningful MCode programs.

3.12.1 Parameters

```
PARAMETERS parameterArray
```

The `PARAMETERS` instruction will provide the visualization engine a list of the types of the parameters used in the method declaration.

```
P ir loc
```

This instruction declares a new parameters in a method call.

3.12.2 Method Declaration

```
MD ir loc
```

A method declaration instruction will indicate the location of the called method code and the beginning of its evaluation.

3.12.3 Static Method Call

```
SMC name declaringClassName numArgs loc
```

The `SMC` instruction consists of the actual name of the method, its declaring class name and the number of arguments this method call consists of.

```
SMCC
```

This instruction indicates the end of a call to a static method.

3.12.4 Object Method Call

```
OMC name numArgs objectReference loc
```

The `OMC` instruction is similar to the static method one. However it contains a reference to an object, this reference points to the instruction that holds the variable access to the object that is making the method call.

```
OMCC
```

As before it just closes the object method call.

3.12.5 Class Allocation

```
SA ir declaringClass constructorName numArgs loc
```

The class allocation instruction happens every time a `new` appears on the source code. It will provide the information needed for the constructor: declaring class, constructor name and the number of arguments it is being called with.

```
SAC ir ObjectId loc
```

3.12.6 Object Field Access

```
OFA ir objectReference name value type loc
```

This instruction accesses to the value of an object field, thus it references to the instruction that evaluate the object (objectReference) (normally a Qualified Name (`QN` one). It contains the name, the type and the value of the accessed field.

3.12.7 Return

```
R callIr loc
```

This first return instruction is just used for the "void" return, those that does not return anything. The `callIr` is the reference for the return value of the method and thus connect the return command to the specific method.

```
R callIr valueIr value type loc
```

This second instruction is used when a value is returned. Thus, we need a reference to the evaluation of the expression corresponding to the returned value (`valueIr`) and the result of this expression, its value and type.

Bibliography

Moreno, A., Jun. 2004. Taxonomy of Intermediate Codes Used in Visualization and its Aplication in Jeliot 3. Master’s thesis, University of Joensuu, Computer Science Department, andres’ thesis.

Myller, N., Mar. 2004. The Fundamental Design Issues of Jeliot 3. Master’s thesis, University of Joensuu, Computer Science Department, niko’s thesis.

Appendix A

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.